Drop Shadows in react-native-windows - Github

Tuesday, July 9, 2019 3:29 PM

General

React Native supports shadows for View, Text, and Image using the ViewStyle.shadow* and TextStyle.textShadow* props. Android and iOS both support these properties (though with some notable differences), but support hasn't been implemented for react-native-windows yet. The WPF RN project has support but a straight port to UWP isn't possible due to differences in shadow APIs available between WPF and UWP + XAML + Composition. There was at least one failed attempt to implement RN shadow support in UWP.

Summary

In my opinion, the iOS properties are the best developer API, in terms of ease of use, intuitiveness, and flexibility. iOS's implementation is also visually superior to Android and Web's shadows. Between color, blur radius, shadow direction (offset), and opacity, everything a developer or designer could want is there. It makes sense to start with those properties.

Proposal

Implement support for ViewStyle.shadow* and TextStyle.textShadow* for View, Text, and Image. In the future, we always have to option of additionally supporting the Android elevation "shadow". Offering the superset could be great for cross-plat apps

Tasks

- A C++/winrt port of `DropShadowPanel`, to be called **`ShadowedContentControl`** will be used for shadows cast from all XAML-based components.
  - This component could optionally be exported so apps authoring their own NativeComponents based on XAML elements can easily add shadow support as well.
- Implement TextStyle.textShadow* support
  - TextViewManager uses a XAML <TextBlock> which can be wrapped in `ShadowedContentControl`
- Implement ViewStyle.shadow* support
  - Short-term: ViewViewManager currently implements ViewStyle.border* and ViewStyle.backgroundColor using a XAML <Border>. This could be wrapped in a DropShadowPanel. [Short-term solution until ViewViewManager can be rewritten to use Composition -- ShapeVisual + DropShadow can be used to render everything without additional XAML elements, and in fact can help reduce ViewViewManager's footprint.
  - Long-term: First rewrite ViewViewManager to draw border visuals using Composition instead of using a XAML <Border>. Once the visuals are drawn using Composition, adding a Composition DropShadow is trivial.
- Implement ViewStyle.sahdow* support for RN <Image>, ImageViewManager
  - ImageViewManager now renders using Composition (via ReactImage and ReactImageBrush) rather than by using a XAML <Image>. It should be possible to integrate DropShadow at the composition level, but we'll need to be careful with any current or future ReactImageBrush caching logic; we wouldn't want the same image asset referenced by two different <Image> components, one with a shadow and one without, to share the same ReactImageBrush, thus having one <Image>'s shadow props affect the other's.
  - Alternatively, the ReactImage could be wrapped using `ShadowedContentControl`. This alleviates concerns over ReactImageBrush caching at the expense of more XAML elements.
- Optionally, extend and expose `ShadowedContentControl` for reuse by third parties
  - Third parties may want to build NativeComponents using XAML elements and respect ViewStyle.shadow* props.
  - `ShadowedContentControl` could be made public in the same way that FrameworkElementViewManager is exposed for external consumption.
  - `ShadowedContentControl` should be extended to support dropping a shadow for arbitrary XAML content by implementing a generic SoftwareBitmap -> AlphaMask Composition helper.

Current Shadow Support Reference

		RN Features				
		ViewStyle. shadow*	TextStyle. textShadowOffset textShadowColor textShadowRadius	ViewStyle.Elevation	TextStyle inherited ViewStyle.shadow* props?	GlyphStyle inherited ViewStyle.shadow* props
RN Implementations	C# WPF	✓	?	✗	?	N/A?
	C# UWP	✗	?	✗	?	N/A?
	C++ WPF	✗	?	✗	?	N/A?
	C++ UWP	✗	✗	✗	✗	✗
	Android	✗	✓	✓(but rendering looks wrong)	✗	N/A
	iOS	✓	✓	✗	✓(but looks different than using identical values on .textShadow* properties)	N/A

Further notes

- Android vs iOS: Android's "elevation" doesn't offer control over color, blur radius, shadow direction or opacity. For system -wide consistency, this could be seen as a positive, but limits what an unconstrainted & talented designer can accomplish.

- Regarding inheritance of ViewStyle.shadow* properties, there are two variants to consider.
  - The first is for components whose Style type inherits from ViewStyle. Should that particular component's implementation also support those shadow properties? TextStyle and ImageStyle both inherit from ViewStyle. TextStyle additionally adds three more textShadow* properties which are preferred. It seems like an accident that the ViewStyle.shadow* props are exposed on TextStyle. Somewhat expectedly, they don't do anything on Android. Surprisingly, on iOS they are respected but produce visually inferior results to using textShadow* instead. If I had to guess, these properties are only accidentally exposed as an implementation detail, and only accidentally supported on iOS because it would be more work to explicitly disable them for Text. Image, as a building block of RN, makes sense to expose and respect ViewStyle.shadow* props on both Android and iOS.
  - The second is whether or not children of a View with a shadow, should implicitly cast a shadow as well.
    - To this point, there are three good reasons that a parent View's shadow props shouldn't be inherited by its children implicit ly:
      - Given that Text shadows are more optimized and visually superior to a Text component using ViewStyle shadow props, it makes sense to make shadows an opt-in experience. This gives the developer the most freedom to optionally render both the view and Text with a shadow, neither, or one, or the other. And in the case that both View and Text have a shadow, the developer may use the more optimized Text shadow with the Text component, thus getting the best performance and visuals. As a corollary, if shadows cascaded then a Text child of a View wouldn't be able to take advantage of the more optimized textShadow* props because it would result in a double-shadow.
      - Some Component types don't support shadows. If the expectation was for shadows to implicitly cascade but some component types don't support shadows, that leads to an inconsistent and broken feeling experience. It's more preferable to have shadows be opt-in, and to have types which don't support them simply not expose the shadow props.
      - If shadows do cascade, how might a developer opt-out on per-child basis? The ViewStyle.shadow* props themselves aren't inherited so one can't simply set style={{shadowColor: undefined}} to opt-out. One could set them to a value that guarantees they don't show, like shadowColor: 'transparent' or shadowOpacity: 0, but that's not ideal.
- iOS's ViewStyle shadows work really well with view border including dotted and dashed border styles. Android treats every vie w as a rectangle (regardless of cornerRadius and whether or not a given border is visible (has a non-zero width and non-transparent color).

UWP has many ways to render shadows, but all of the standard approaches are limited to only TextBlock, Image, and Shape. With some exceptions.

Absolute Positioning

Position: 'absolute' should be respected and the shadow should still be positioned relative to the content that is casting it. I'd assume this will just work, with both ShadowedContentControl and direct use of Composition DropShadow.

Clipping

Clipping the shadow in accordance to how source content is clipped (for both animating and normal scenarios) is also important for visual consistency.

Text clipping (overflow: none) should also trim the shadow. A long appointment text that is clipped shouldn't cast a shadow that falls outside the appointment nor should the shadow reveal more text than is visible. TextBlock.ActualWidth doesn't take clipping into account so we'll need to size the shadow another way, if TextBlock.GetAlphaMask doesn't already account for this.

RNWCPP-specific Components

Glyph.uwp.tsx uses NativeComponent 'PLYIcon', implemented by IconViewManager which uses XAML 'Glyphs' control. This control doesn't support shadows. One option might be to reimplement IconViewManager using Composition or using a XAML TextBlock. Alternatively, we could generate the AlphaMask using a Composition rush from a SoftwareBitmap.

Implementations

	Library	Language	Supported Controls	Other features	Performance	MinVersion
ThemeShadow	UWP	C++	TextBlock, Image, Shape	Adapts to lighting, theme, elevation, **syncs with animations?** Consistent across applications	Best? Used by many built-in XAML controls	18362 (19H1)
Composition DropShadow	UWP Composition	C++	Composition SpriteVisual and	Most flexible ✓Supports animations	Best, but varies depending on usage.	14393 (RS1) "SourcePolicy" 16299 (RS3)
DropShadowPanel documentation source	Windows Community Toolkit	C#	TextBlock, Image, Shape			
Mtaulty blog	Custom		Everything!		Worst	

ThemeShadow

Of the pre-built solutions, ThemeShadow is the most robust as it provides support for lighting. It doesn't allow control over the color. I believe all ThemeShadows in an application (or even system-wide) must be cast from the same light source -- no control over direction.

DropShadowPanel

Is implemented only in C#, so we won't be taking a dependency on Windows Community Toolkit. However, the implementation is open source and might be worth rewriting in c++/winrt. The DropShadowPanel is actually a misnomer, it is simply a component derived from ContentControl. It is only capable of adding a shadow to a single child. However, if combined with the Composition SoftwareBitmap AlphaMask technique, this could include the View's panel and all of its children (asking to how iOS' ViewStyle.shadow* support appears to apply to all children as a matter of implementation detail, rather than because the choice was preferred.)

React Native Shadows

Whenever possible, we prefer to support existing React Native features in react-native-windows instead of creating windows-only React Native properties. Below are the shadow features RN currently has

TextStyle.textShadowRadius: number
ViewStyle.elevation (android only)
ViewStyle.shadowColor, shadowOffset, shadowOpacity, shadowRadius

WPF has easier to use shadow APIs than UWP, Composition, and XAML offer. React-native-windows supports shadows for WPF, but not for UWP C# nor UWP C++. There was one failed UWP C# implementation attempt, but it wasn't allowed to merge due to fragility of implementation. As a reference, that implementation is here: https://github.com/microsoft/react-native-windows/pull/1633/files

Open Issues

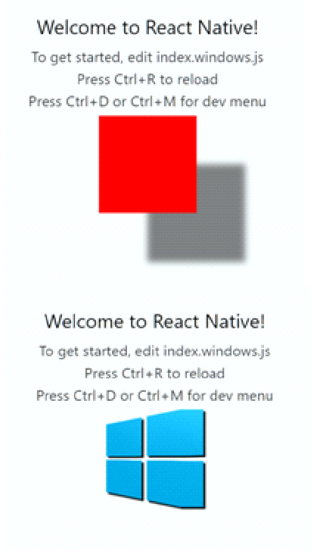
1. What about absolute positioning? Will shadows work as expected?

Further reading

- https://review.docs.microsoft.com/en-us/windows/uwp/design/layout/depth-shadow
- https://docs.microsoft.com/en-us/windows/uwp/composition/composition-shadows
- Code
  - DropShadowPanel https://github.com/windows-toolkit/WindowsCommunityToolkit/blob/master/Microsoft.Toolkit.Uwp.UI.Controls/DropShadowPanel/DropShadowPanel.cs
  - SoftwareBitmap -> CompositionBrush -> AlphaMask Sample code here (scroll down to "and this chunk of code-behind;")
  - RNW vNext ReactImage, ReactImageBrush, https://github.com/microsoft/react-native-windows/tree/master/vnext/ReactUWP/Views/Image

Prototyping & Exploration

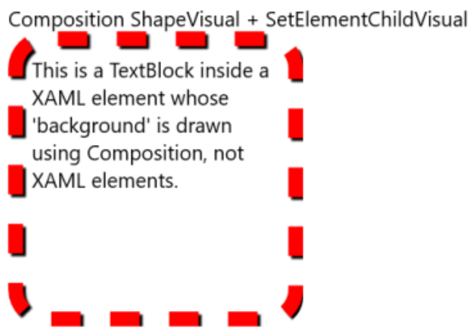
View shadow and image shadow examples from unmerged RNW C# contribution


Welcome to React Native!
To get started, edit index.windows.js
Press Ctrl+R to reload
Press Ctrl+D or Ctrl+M for dev menu

Welcome to React Native!
To get started, edit index.windows.js
Press Ctrl+R to reload
Press Ctrl+D or Ctrl+M for dev menu

[Rob's prototype] Honoring border* and shadow* properties together

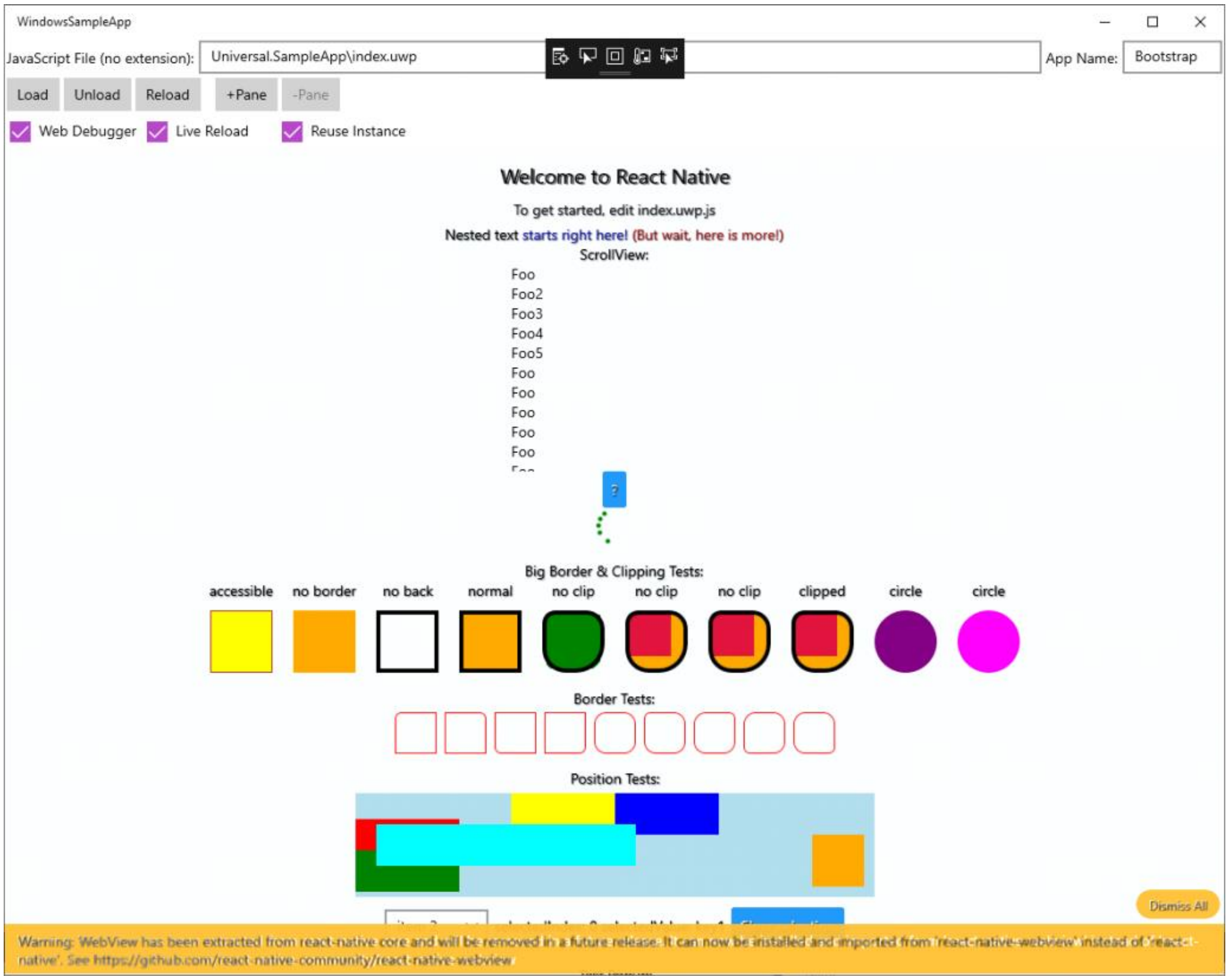
RNW vnext currently uses a XAML Border control to render View.border* prop visuals. There are a lot of downsides


Composition ShapeVisual + SetElementChildVisual

This is a TextBlock inside a XAML element whose 'background' is drawn using Composition, not XAML elements.

[Rob's prototype] Text Shadows sort-of working in RNWCPP

Shadows here are drawn above the TextBlock because Composition doesn't allow inserting a visual anywhere but at the top of the Z order. This will be resolved by ShadowedContentControl


WindowsSampleApp

JavaScript File (no extension): Universal.SampleApp\index.uwp

App Name: Bootstrap

Web Debugger ✓ Live Reload ✓ Reuse Instance ✓

Welcome to React Native

To get started, edit index.uwp.js

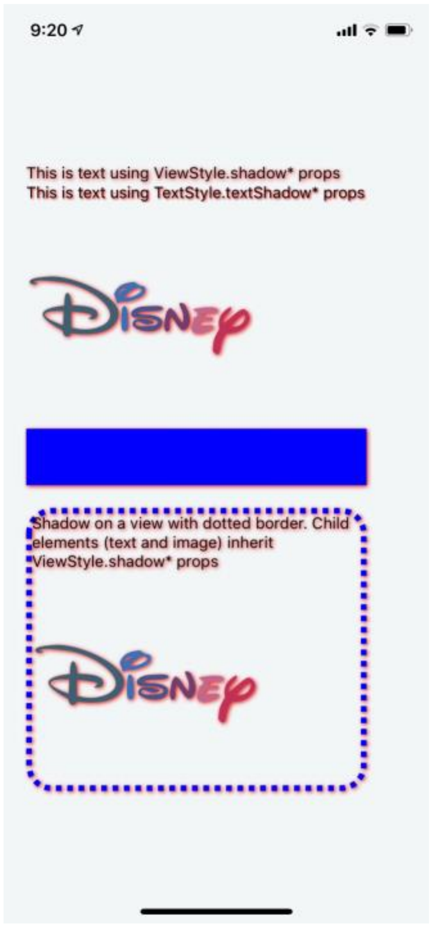
Nested text starts right here! (But wait, here is more!)

ScrollView:

Foo
Foo2
Foo3
Foo4
Foo5
Foo
Foo
Foo
Foo
Foo

Warning: WebView has been extracted from react-native core and will be removed in a future release. It can now be installed and imported from 'react-native-webview' instead of 'react-native'. See https://github.com/react-native-community/react-native-webview

[Rob's prototype] RN iOS Shadow gallery


9:20
This is text using ViewStyle.shadow* props
This is text using TextStyle.textShadow* props
Shadow on a view with dotted border. Child elements (text and image) inherit ViewStyle.shadow* props

```
import * as React from 'react';
import { Image, Text, View, StyleSheet } from 'react-native';
import Constants from 'expo-constants';

// You can import from local files
import AssetExample from './components/AssetExample';

// or any pure javascript modules available in npm
import { Card } from 'react-native-paper';

export default class App extends React.Component {
  render() {
    return (
      <View style={styles.container}>

        <Text style={{shadowColor: "red", shadowRadius: 2, shadowOpacity: 1, shadowOffset: {width: 1, height: 1}}}>This is text using ViewStyle.shadow* props</Text>
        <Text style={{textShadowColor: 'red', textShadowRadius: 2, textShadowOffset: {width: 1, height: 1}}}>This is text using TextStyle.textShadow* props</Text>

        <Image source={{uri: 'https://cdn.freebiesupply.com/logos/large/2x/disney-1-logo-png-transparent.png'}} style={{width: 200, height: 200, shadowColor: "red",
shadowRadius: 2, shadowOpacity: 1, shadowOffset: {width: 1, height: 1}}} />

        <View style={{width: 300, height: 50, backgroundColor: "blue", shadowColor: "red", shadowRadius: 2, shadowOpacity: 1, shadowOffset: {width: 1, height: 1},
marginBottom: 20}} />

        <View style={{width: 300, height: 250, borderColor: "blue", borderWidth: 5, borderRadius: 20, borderStyle: 'dotted', shadowColor: "red", shadowRadius: 2,
shadowOpacity: 1, shadowOffset: {width: 1, height: 1}}}>
          <Text>Shadow on a view with dotted border. Child elements (text and image) inherit ViewStyle.shadow* props</Text>
          <Image source={{uri: 'https://cdn.freebiesupply.com/logos/large/2x/disney-1-logo-png-transparent.png'}} style={{width: 200, height: 200}} />
        </View>

      </View>
    );
  }
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    paddingTop: Constants.statusBarHeight,
    backgroundColor: '#ecf0f1',
    padding: 20,
  },
});
```